RESUMO UNIDADES 3 E 4 DE COMPILADORES

INTRODUÇÃO

A construção de um compilador é um processo complexo dividido em duas grandes fases: a análise, frequentemente denominada *front-end*, e a síntese, conhecida como *back-end*. A fase de análise é responsável por verificar se o código-fonte está escrito de acordo com as regras definidas pela gramática da linguagem. Já a fase de síntese tem como objetivo transformar o código, que passou pela análise, em código alvo (executável).

O processo de compilação inicia com a análise léxica, seguida pela análise sintática e, por fim, pela análise semântica. A fase de síntese, por sua vez, divide-se em geração de código intermediário, otimização do código e geração do código alvo. Todas essas etapas, desde a análise até a síntese, utilizam uma estrutura de dados fundamental: a **tabela de símbolos** (TS).

Este resumo tem como objetivo detalhar as partes mais importantes envolvidas na construção de um compilador, focando nas etapas críticas de análise (sintática e semântica) e nos processos de geração de código, otimização e a importância central da Tabela de Símbolos, conforme o material de estudo.

FASE DE ANÁLISE (FRONT-END)

A fase de análise verifica se o arquivo-fonte foi escrito em conformidade com as regras da linguagem. Dentro do *front-end*, a **análise sintática** é considerada o "coração da análise" e é essencial para o desenvolvimento do compilador.

Análise Sintática: O Coração da Análise

A sintaxe de uma linguagem define as estruturas básicas, que são compostas por regras léxicas (definidas por gramáticas regulares - GR) e regras sintáticas (definidas por gramáticas livres de contexto - GLC). O analisador léxico é responsável por ler o código-fonte e gerar *tokens* válidos; em seguida, o analisador sintático verifica se a sequência de *tokens* recebida está na ordem correta, seguindo as regras sintáticas.

A função do analisador sintático é agrupar *tokens* em uma "frase gramatical" e gerar uma representação hierárquica, conhecida como a **árvore gramatical** ou árvore de derivação. Se a derivação da árvore sintática for concluída, alcançando os símbolos terminais (folhas), a sentença (estrutura de comando) pertence à linguagem.

Métodos de Análise Sintática

Existem dois métodos principais para a análise sintática:

1. **Descendente (*Top-down*)**: Analisa a árvore gramatical da raiz para as folhas. Este método é considerado o mais intuitivo e usual.
2. **Ascendente (*Bottom-up*)**: Constrói a árvore gramatical das folhas para a raiz. O CUP (Constructor of Useful Parsers) implementa um analisador LALR (*Look-Ahead Left-to-Right*), que é um analisador ascendente com análise preditiva.

Os analisadores descendentes geralmente usam um **modelo de algoritmo preditivo recursivo**. Para ser eficiente, o analisador *top-down* ideal deve operar **sem retrocesso** (backtracking), pois o retrocesso é ineficiente em termos de tempo e recuperação de erros.

Desafios da Gramática na Análise Sintática

Para que um analisador *top-down* preditivo recursivo sem retrocesso possa ser utilizado, duas condições cruciais devem ser atendidas pela gramática:

1. **Ausência de Ambiguidade:** Uma gramática é dita ambígua se produzir mais de uma árvore gramatical distinta para a mesma sentença. A ambiguidade não é recomendável, especialmente porque não trata corretamente a precedência de operadores, por exemplo. Eliminar a ambiguidade envolve reescrever a gramática, muitas vezes construindo mais produções para tratar operadores distintamente.
2. **Ausência de Recursão à Esquerda:** O algoritmo *top-down* sem retrocesso exige que a gramática não tenha recursão à esquerda. A recursão à esquerda ocorre quando uma produção à esquerda aparece também à direita, gerando um processo recursivo. Para eliminar a recursividade à esquerda, é necessário **fatorar** a gramática, deslocando a recursão para a direita.

O analisador sintático também possui a função de sinalizar erros de sintaxe. A **recuperação de erros** é fundamental para a análise completa do código-fonte após a detecção de um erro. Estratégias de recuperação de erros incluem a Modalidade do Desespero, Nível de Frase, Correções de Erros e Correção Global.

Análise Semântica e Tradução Dirigida pela Sintaxe

A análise semântica constitui a última fase da análise. Ela é o processo de analisar o contexto da frase (ou comandos de programação) para verificar se estão corretos. A análise semântica conecta as definições das variáveis com seu uso, verifica a compatibilidade dos tipos em expressões e traduz a sintaxe abstrata em uma representação adequada para a geração de código.

Os aspectos semânticos são tratados por sub-rotinas acionadas pelo analisador sintático. A técnica utilizada para associar esses aspectos semânticos à gramática é a **tradução dirigida pela sintaxe**.

Notações e Atributos Semânticos

A tradução dirigida pela sintaxe utiliza duas notações:

1. **Definições dirigidas pela sintaxe (DDS):** Especificações de alto nível onde um conjunto de atributos é associado a cada *token*.
2. **Esquemas de tradução:** As ações semânticas são inseridas no lado direito das produções, permitindo determinar a ordem de avaliação dos atributos.

Os atributos, associados aos símbolos da gramática, podem ser:

- **Sintetizados:** A avaliação da regra semântica é feita de baixo para cima (da folha para a raiz). O analisador léxico, por exemplo, fornece o atributo sintetizado (*lexval*) para os *tokens*.
- **Herdados:** O valor semântico do nó é definido pelo nó superior (pai) e pelo nó lateral (irmão).

Uma definição dirigida pela sintaxe que utiliza **somente atributos sintetizados** é chamada de **definição S-atribuída**.

A análise semântica deve tratar erros de:

- Declaração de identificadores (variáveis não declaradas).
- Compatibilidade de tipos.
- Compatibilidade entre parâmetros e argumentos.

A Tabela de Símbolos (TS)

A Tabela de Símbolos (TS) é um elemento auxiliar presente em **todas as fases** do compilador (léxica, sintática, e semântica), além de ser usada na fase de síntese (*back-end*). A TS armazena informações sobre os *tokens* e seus atributos.

A TS começa a ser preenchida no analisador léxico, que insere o par (*token*, *lexema*). Na análise sintática, a TS é consultada para acrescentar ou alterar atributos, ou incluir novos elementos.

A implementação da TS é crucial devido a fatores como:

- O grande número de elementos a serem inseridos (milhares de *tokens* em programas longos).
- A diversidade na estrutura dos elementos (variáveis, tipos, funções, etc., que possuem diferentes atributos).
- A enorme quantidade de operações de busca, inclusão e alteração, exigindo **resposta rápida** para que o processo de compilação não seja lento.

Portanto, a implementação da TS utilizando **tabela *hash*** (*Hashtable*) confere a melhor eficiência ao processo de compilação. O princípio da tabela *hash* é permitir a busca direta em tempo essencialmente constante ($\sigma(1)$). Para isso, é necessário definir uma função *hash* e um tratamento de colisão.

FASE DE SÍNTESE (BACK-END)

A fase de síntese é responsável por transformar a representação da análise (árvore sintática anotada) no código alvo, que é o código executável pela máquina. Esta fase compreende a geração do código intermediário, a otimização e a geração do código alvo.

Geração de Código Intermediário (RI)

A geração de um **código intermediário** (RI), que é independente da arquitetura da máquina, é uma solução moderna e altamente vantajosa.

Vantagens da Representação Intermediária (RI)

A RI simplifica significativamente a construção de compiladores portáveis:

1. **Portabilidade:** Ao criar uma RI, não é necessário reescrever um compilador para traduzir diretamente a Árvore Sintática Abstrata (AST) para o código de máquina, um processo complexo. Se houver diversas plataformas, apenas a tradução do código intermediário para o código final precisará ser feita, tornando o processo mais simples.
2. **Otimização:** A RI permite uma análise muito melhor na etapa mais complexa do compilador, a otimização do código, antes da geração do código final.

Formas de Representação Intermediária

Segundo Aho (2007), há três formas básicas de representação intermediária:

1. **Árvores Sintáticas Abstratas (AST):** A árvore é armazenada em uma estrutura de dados, onde cada nó é representado por um registro.

2. **Notação Pós-fixa:** Uma representação linear, também chamada de notação polonesa reversa. Essa notação utiliza o conceito de pilha, onde a operação é inserida após as variáveis ou números (exemplo: $A \ B \ C + * D /$). É útil para expressões numéricas e simplifica a conversão de instruções complexas.
3. **Código de Três Endereços:** As instruções complexas são convertidas em uma estrutura intermediária simples que utiliza apenas três endereços. Essa forma facilita a otimização e a geração do código final. O *bytecode* da JVM (Java Virtual Machine) é um exemplo de código intermediário.

Otimização do Código

A fase de otimização visa gerar um código alvo eficiente, que não seja redundante e utilize plenamente os recursos do processador. Embora não exista um método que garanta o código *ótimo*, é possível garantir um código *bom* com a aplicação de técnicas de otimização.

As transformações de otimização devem seguir três propriedades:

1. **Preservação do Significado:** A otimização não pode comprometer a execução do programa nem gerar erros que não existiam no código fonte original.
2. **Aceleração Mensurável:** A transformação deve acelerar o programa, em média, por um fator mensurável.
3. **Compensação de Esforço:** O tempo gasto pelo compilador na otimização deve ser compensado pela eficiência do programa gerado.

A otimização frequentemente utiliza a decomposição do código intermediário (três endereços) em **blocos básicos (BB)** e a construção de **grafos de fluxo de controle (CFG)** para análise. Um bloco básico é uma sequência de enunciados consecutivos onde o controle entra no início e sai no fim, sem ramificações internas. A identificação de BBs, que começam com um "líder" (o primeiro enunciado, o alvo de um desvio, ou o enunciado que sucede um desvio), é fundamental para a sistematização da otimização. Estruturas recursivas, como loops (*while*), são candidatas frequentes à otimização.

Geração de Código Alvo

A fase de geração de código converte a Representação Intermediária (RI) em um código que possa ser executado na máquina de destino (código alvo). Essa fase é desenvolvida em três ações principais, que reescrevem o código intermediário para refletir as decisões tomadas:

1. **Seleção de Instruções:** Escolher uma sequência de instruções da máquina-alvo que implemente as operações da RI.
2. **Escalonamento das Instruções:** Definir a ordem na qual as operações devem ser executadas.
3. **Alocação de Registros:** Definir quais valores devem residir nos registradores em cada ponto do programa.

O exemplo do *bytecode* JVM mostra a geração de código alvo (seleção de instruções), onde as instruções RI de três endereços são convertidas para o conjunto de instruções de máquina (bytecode).

CONCLUSÃO

A construção de um compilador requer a integração de diversos fundamentos teóricos da computação, incluindo linguagens formais, estrutura de dados e algoritmos. A fase de análise estabelece a validade estrutural e contextual do código, sendo a análise sintática o ponto central, tratando a estrutura da linguagem por meio de GLCs. A análise semântica, auxiliada pela **tradução dirigida pela sintaxe**, garante a coerência lógica e a compatibilidade de tipos.

A **Tabela de Símbolos**, implementada de forma eficiente por **tabelas *hash***, é o eixo de suporte que viabiliza a comunicação e o armazenamento de atributos entre todas as etapas do *front-end* e *back-end*.

Na fase de síntese, a **Representação Intermediária (RI)** — seja por AST, notação pós-fixa ou código de três endereços — é crucial para garantir a portabilidade do compilador e facilitar o processo de otimização. Por fim, a geração de código alvo, que envolve seleção, escalonamento e alocação de registros, transforma o código otimizado em instruções de máquina. Dominar essas fases essenciais é fundamental para qualquer profissional que busque excelência na área de desenvolvimento de software de baixo nível e de linguagens de programação. 💡

REFERÊNCIAS

AHO, A. V.; SETHI, R.; ULLMAN, J.D. Compiladores: Princípios, técnicas e ferramentas. 2ª ed. São Paulo: Pearson, 2007.

APPEL, A. W. Modern Compiler Implementation in Java. 2. ed. Cambridge University Press, 2002.